一、什么是网络编程

2024年6月26日　　14:41

一、什么是网络编程

1.1 网络编程的基本概念和目的

进程间的通信：管道（pipe、fifo）、内存映射（mmap）、消息队列（mqueue）

局限性：只能作用于一个系统（一台计算之间）

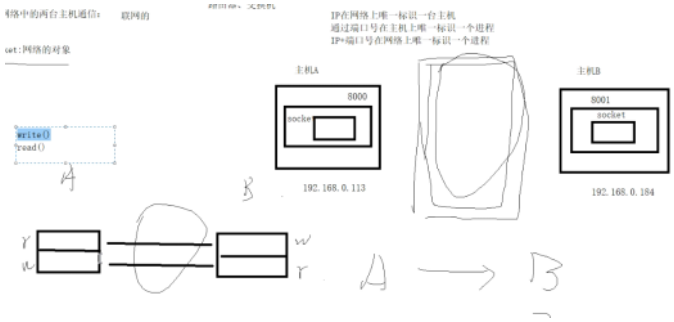
socket（套接字）：进程间的通信（一台主机的进程、多台主机之间的进程通信）

两台主机之间通信的要求：联网、两台主机之间有物理连接（无线网（路由器）、网线（交换机））

通过IP地址，可以在网络中唯一标识一台主机
通过端口号，可以在主机上唯一标识一个进程

通过IP地址+端口号，就可以在网络上的不同主机间，唯一标识一个进程

socket有两个缓冲区（读缓冲区、写缓冲区）



在网络通信的过程中，一般将上面的两端分为 客户端（C）和服务器(B)

网络编程是指在计算机程序中实现网络通信和数据交换的过程。它涉及到使用编程语言和网络协议来实现不同计算机之间的数据传输、通信和交互。网络编程的基本概念和目的包括以下内容

基本概念:
1. 套接字(socket):
套接字是网络编程的核心概念，它提供了一个接口，允许程序在网络上发送和接收数据。套接字允许应用程序与网络之间建立连接并进行通信。

2. 协议:
网络通信需要遵循一定的规则和约定，这些规则称为协议。常见的网络协议如TCP、UDP、HTTP等，规定了数据的传输方式、格式和流程。

3. IP 地址和端口号:
IP 地址用于标识计算机或设备在网络中的位置，端口号用于标识进程在计算机上的通信端口。组合起来，它们实现了源和目标主机之间的通信。客户端和服务器: 在网络编程中，通常有客户端和服务器两种角色。客户端发送请求，服务器接收请求并提供响应。这种模式实现了分布式计算和资源共享。

目的:
数据传输和通信: 网络编程使得不同计算机之间可以传输各种类型的数据，包括文本、图像、音频、视频等，实现

信息传递和通信。

数据交换和合作：网络编程支持不同应用之间的数据交换和合作，使得不同系统能够相互通信，从而实现更复杂的功能。

1.2 网络编程的应用场景

网络编程的应用场景非常广泛，涵盖了多个领域，从互联网到物联网，从嵌入式系统到大型分布式应用。以下是一些常见的网络编程应用场景：

1. 网站和 Web 应用：网络编程用于构建网站和 Web 应用，实现用户与服务器之间的数据交换和通信。这包括前端与后端的通信，用户提交表单、查看内容、进行在线购物等功能
2. 实时通信：聊天应用、即时消息、社交媒体等应用都依赖网络编程来实现实时通信功能使得用户能够即时交流，
3. 远程访问：通过网络编程，用户可以远程访问其他计算机上的文件、应用、数据库等资源，实现远程工作、远程控制等功能。:大型分布式系统，如云计算平台、分布式数据库、分布式存储系统等，都需
4. 分布式系统:要网络编程来实现计算资源的分布和协调。
5. 物联网(IoT)：物联网设备之间的通信和数据交换依赖于网络编程。从智能家居到工业自动化，网络编程在物联网中发挥着重要作用。
6. 在线游戏：多人在线游戏(MMOG)和多人在线角色扮演游戏(MMORPG)等游戏需要网络编程来实现玩家之间的实时互动。
7. 实时视频和音频通话：视频会议、语音通话应用以及直播平台都需要网络编程来支持实时的视频和音频传输。
8. 电子商务：网络编程在电子商务平台中用于实现用户下订单、支付、物流追踪等功能。
9. 数据传输和存储：网络编程用于将大量数据传输到不同地点，也用于构建分布式存储系统。
10. 远程监控和控制:在工业自动化、安防系统等领域，网络编程可以实现远程设备的监控和控制。
11. 金融交易：网络编程在金融领域用于实现在线银行、股票交易平台等，确保安全的数据传输和交易操作。

总之，网络编程在现代计算机应用中扮演着关键角色，它使得不同计算机之间能够实现数据传输、通信和交互，从而支持各种各样的应用场景，提升了用户体验、便捷性和效率。

1.3 网络编程的特点优势

Linux下的网络编程具有许多特点和优势，这使得它成为开发网络应用的首选平台之一。以下是linux下网络编程的一些特点和优势

开源性：linux是开源操作系统，拥有庞大的开发社区支持。开源性使得开发者可以自由获取、修改和分发网络编程相关的工具和库，

丰富的网络编程工具：linux提供了丰富的网络编程工具和库，如套接字(套接字)、网络协议栈(tcp/ip、udp)、网络调试工具(如wireshark)等，使得开发者可以更方便地实现各种网络应用.

稳定性和可靠性：linux以其稳定性和可靠性而闻名，这对于需要长时间运行的服务器应用尤为重要.它具备良好的内存管理、进程管理和错误处理机制.

广泛的硬件支持：linux支持广泛的硬件设备和体系结构，从嵌入式设备到大型服务器，都可以运行linux，使得网络应用可以适应不同的硬件环境。

多任务和多线程支持：linux支持多任务和多线程，允许开发者实现同时处理多个连接和请求的网络应用，提高了性能和并发性，

命令行和脚本支持：linux的命令行界面和丰富的脚本语言支持(如Bash、Python等)使得网络管理和编程变得更加灵活和高效.

性能优化：linux针对网络应用进行了许多性能优化，如零拷贝技术、多路复用(复用)等，提高了网络应用的效率

安全性：linux提供了强大的安全性特性，包括访问控制、用户权限、防火墙等，有助于保护网络应用免受恶意攻击.

社区支持和文档丰富：linux拥有庞大的开发者社区和丰富的文档资源，开发者可以在社区中获取帮助、分享经验和解决问题.

二、创建网络应用程序

2024年6月26日 17:24

二、创建网络应用程序

2.1 "B/S"和"C/S"模式

"C/S" 和"B/S"是两种常见的软件架构模式，用于描述客户端和服务器之间的关系。它们分别代表"Client/Server"(客户端/服务器)和"Browser/Server"(浏览器/服务器)。

C/s(Client/server)模式:

在 C/S 模式中，应用程序被分成两部分:客户端和服务器。客户端是指运行在用户计算机上的应用程序，用户通过客户端与服务器进行交互。服务器是指运行在服务器计算机上的应用程序，负责处理客户端的请求并提供相应的服务。C/S 模式可以实现复杂的逻辑和功能，客户端可以是桌面应用程序、移动应用程序等。

B/S(Browser/server)模式:

在 B/S 模式中，应用程序逻辑主要运行在服务器上，而客户端通常是一个 Web 浏览器。用户通过浏览器访问网页，浏览器向服务器发出请求，服务器处理请求并将网页内容传送给浏览器进行显示。这种模式下，用户不需要安装额外的客户端软件，只需要有一个支持 Web 浏览的设备。

区别和特点:

部署方式: C/S 模式需要在每个用户设备上安装客户端软件，而 B/S 模式只需要一个普通的Web 浏览器。
更新和维护: C/S 模式需要在每个客户端上进行软件更新，而 B/S 模式的更新只需在服务器上进行。
跨平台性: B/S 模式更具有跨平台特性，因为Web 浏览器几乎在所有操作系统上都可用
资源消耗: C/S 模式在客户端上消耗较多的系统资源，而 B/S 模式减轻了客户端的负担。
安全性: B/S 模式通过 Web 浏览器传输数据，可以使用 HTTPS 等协议来提供更高的安全性

通常情况下，选择 C/S 还是 B/S 模式取决于应用的特性和需求。复杂的桌面应用程序可能选择 C/S 模式而需要跨平台和易于维护的应用程序可能更适合 B/S 模式。

2.2 Socket 编程简介

Socket(套接字)是在网络编程中用于实现不同计算机之间通信的接口和抽象。它允许应用程序通过网络发送和接收数据，实现网络通信。
前面我们了解到了编写一个网络程序如果遵循 TCP/IP 四层模型的话我们应该对每一层都要进行处理，网络层使用 IP 协议、传输层使用 TCP 或 UDP 协议等等，这些真的需要我们自己进行处理嘛，当然不是，这要是让你处理就你那逼样的这辈子也学不明白网络编程了，这些底层的协议处理已经集成在你的系统中了，我们只需要通过一个接口去使用即可，至于底层的代码究竟是如何实现的完全不用你操心，也就是说我们不会直接接触 TCP、UDP、IP 等底层协议。

套接字类型: Socket有不同的类型，常见的有**流式套接字(SOCK_STREAM)**和**数据报套接字(SOCK_DGRAM)**。流式套接字提供了可靠的、面向连接的通信，如 TCP 协议;数据报套接字提供了不可靠的、无连接的通信，如 UDP 协议。
创建套接字: 在编程中，首先需要创建一个套接字。调用socket()函数，指定套接字的域(如AF_INET表示 IPv4)、类型和协议，返回一个套接字描述符,linux 中通过一个系统调用 socket 来创建一个套接字，下面是他的函数原型:

socket()函数介绍

函数描述
创建一个套接字
头文件
```
#include <sys/socket.h>
```
函数原型

```
int socket(int domain, int type, int protocol);
```

函数参数

1、参数 domain

domain是“域”的意思，其值为AF_INET

在Linux系统中，domain参数用于指定套接字的协议域（protocol domain），它定义了套接字通信的协议族。以下是Linux系统中一些常见的domain值：

AF_UNIX：Unix 域协议域，用于本地通信（Inter-process communication，IPC）。它使用文件路径作为套接字地址，用于同一台机器上的进程间通信。

AF_INET：IPv4 协议域，用于 Internet 地址族。这是最常见的协议域，用于基于 IPv4 的网络通信。

AF_INET6：IPv6 协议域，用于 IPv6 地址族。这是用于基于 IPv6 的网络通信。

AF_PACKET：用于原始网络数据包的协议域。它允许应用程序直接访问网络帧，适用于网络协议分析和数据包捕获等场景。

AF_BLUETOOTH：蓝牙协议域，用于蓝牙通信。

AF_X25：X.25 协议域，用于 X.25 网络协议。

AF_NETLINK：Netlink 协议域，用于 Linux 内核与用户空间进程之间的通信。

AF_PACKET：原始数据链路层套接字，允许应用程序直接访问数据链路层帧。

2、参数 type

type指定套接字的类型，可以是以下值之一：

SOCK_STREAM：流套接字，用于可靠、面向连接的服务。对应于 TCP 协议。

SOCK_DGRAM：数据报套接字，用于无连接、不可靠的服务。对应于 UDP 协议。

SOCK_SEQPACKET：顺序数据包套接字，在 SCTP 协议中使用。

SOCK_RAW：原始套接字，用于直接访问底层网络协议。可以自定义协议头部并发送。

SOCK_RDM：可靠数据报套接字，很少使用。

SOCK_PACKET：废弃的套接字类型，已经不再使用。

3、参数protocol

在socket函数中，protocol参数用于指定套接字使用的协议。

协议（protocol）是一组规则和约定，用于在网络中的不同节点之间进行通信和数据交换。

下面是一些常见的protocol参数值及其对应的协议：

IPPROTO_TCP：TCP（Transmission Control Protocol）协议。它是一种面向连接的、可靠的、基于字节流的传输协议，用于提供可靠的数据传输。

IPPROTO_UDP：UDP（User Datagram Protocol）协议。它是一种无连接的、不可靠的、基于数据报的传输协议，用于提供快速的数据传输，但不保证数据的可靠性和顺序性。

IPPROTO_SCTP：SCTP（Stream Control Transmission Protocol）协议。它是一种面向连接的、可靠的、基于消息的传输协议，提供了可靠的数据传输和流量控制等功能。

IPPROTO_ICMP：ICMP（Internet Control Message Protocol）协议。它是一种网络层协议，用于在网络中传递控制信息和错误报文，如网络不可达、请求超时等。

IPPROTO_IGMP：IGMP（Internet Group Management Protocol）协议。它是一种组播协议，用于在 IP 网络中进行组播组的管理和维护。

IPPROTO_RAW：原始 IP 协议。它允许应用程序直接访问网络层的数据，可用于构造和发送自定义的 IP 报文。

需要注意的是，protocol参数的具体取值取决于所选择的协议域（domain）和套接字类型（type）。在某些情况下，可以将protocol设置为0，表示使用默认协议。此时，系统会根据协议域和套接字类型自动选择适合的协议。

参数type和参数protocol之间的关系

一般来说：

SOCK_STREAM 对应 IPPROTO_TCP

SOCK_DGRAM 对应 IPPROTO_UDP

SOCK_SEQPACKET 对应IPPROTO_SCTP

SOCK_RAW 对应IPPROTO_ICMP、IPPROTO_RAW和IPPROTO_IGMP

由此，你可以大概知道当Linux中的socket函数的参数domain和参数type确定后，参数protocol该怎么选

函数返回值

如果没有发生错误，socket将返回一个引用新socket的描述符。

否则，将返回INVALID_SOCKET的值，并且可以通过调用WSAGetLastError检系特定的错误代码。

```
// 创建socket
// 参数
// AF_INET: 套接字协议域为IPV4协议
// SOCT_STREAM: 套接字类型为 流套接字（安全，稳定，不丢包，对应TCP协议）
// 0 : 套接字使用默认协议（流式套接字为TCP，报式套接字为UDP协议）
// TPC协议参数 IPPROTO_TCP
int lfd = socket(AF_INET,SOCK_STREAM,0);  // 创建一个流式套接字
int lfd1 = socket(AF_INET,SOCK_DGRAM,0);  // 创建一个报式套接字
int lfd2 = socket(AF_INET,SOCK_RAW,0);  // 创建一个原始套接字
// 返回一个 socket描述符
// SOCK_STREAM 是流式协议
// SOCK_DGRAM  是报式协议
// TCP是流式的代表
// UDP是报式的代表
```

创建套接字之后，还得将套接字和 IP地址与端口号绑定
套接字绑定 IP 地址和端口号:服务器端需要将套接字与特定的IP地址和端口号绑定起来，使用bind()函数。

bind()函数介绍

函数描述

将套接字与特定的IP地址和端口号绑定

函数原型

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

函数参数

int sockfd： 要绑定的socket描述符

const struct sockaddr *addr： 一个结构体地址，用来做缓冲区（这个结构体已经弃用，用的是下面的这个）

结构体所需的头文件#include <netinet/ip.h>

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */（网络字节序的端口号）
    struct in_addr sin_addr;   /* internet address */存储IP地址
};
```

（1）sin_family指代协议族，在socket编程中只能是AF_INET

（2）sin_port存储端口号（使用网络字节顺序），2个字节（16位，0-65535），端口号一般大于1000，找个没人用的端口号，

需要考虑大小端的问题，也就是网络字节序和主机的大小端是否相同（一般不同）

一般主机的字节序是小端存储、网络字节序是大端存储

因此需要将端口号的存储方式改为网络字节序

使用 htons() 函数

（3）sin_addr存储IP地址，使用in_addr这个数据结构

使用ifconfig 查看IP地址为 192.168.10.128 ，这是一个主机（本地）字节序的字符串

s_addr的类型是uint32_t，应该是一个32位（4个字节的）整型

因此要将 上面 本机字节序的字符串 转换为 一个 网络字节序的整型

使用 inet_pton()函数转换

或者直接传值 INADDR_ANY 这是一个宏定义，值为0，传入之后可以是本机的任意一个IP地址

```
struct in_addr{
        uint32_t  s_addr;     /* address in network byte order */（网络字节序的IP地址）
        };
```

socklen_t addrlen： 结构体缓冲区的长度

```
sizeof(addr);
```

函数返回值

成功返回0

失败返回-1，并记录错误信息

给套接字绑定ip+port

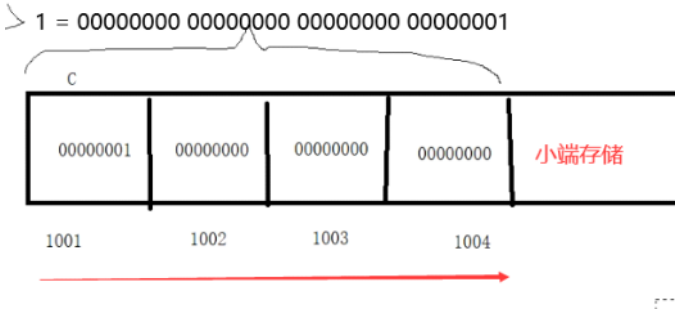
```
// 自定义端口号
// 需要考虑大小端的问题，也就是网络字节序和主机的大小端是否相同（一般不同）
// 一般主机的字节序是小端存储、网络字节序是大端存储
// 因此，需要自己转换字节序
#define SOCKPORT 8001
```

```
// 使用 bind()函数，绑定IP地址、端口号
struct sockaddr_in serAddr;
/*
sockaddr_in 结构体的参数
in_family指代协议族，在socket编程中只能是AF_INET
sin_port存储端口号（使用网络字节顺序），2个字节（16位，0-65535）
sin_addr存储IP地址，使用in_addr这个数据结构
*/
serAddr.sin_family  = AF_INET;
// 端口号一般大于1000，找个没人用的端口号
// 一般主机的字节序是小端存储、网络字节序是大端存储
// 因此，需要自己转换字节序
serAddr.sin_port = htons(SOCKPORT);
// htons 将主机字节序转为网络字节序（短整型）
// host to network short
// 主机 向  网络    短整型
// ip 地址 192.168.10.128 --- 主机（本地）字节序的字符串
// uint32_t 应该是一个32位（4个字节的）整型
// 因此要将 上面 本机字节序的字符串 转换为 一个 网络字节序的整型
serAddr.sin_addr.s_addr = INADDR_ANY;
// 使用 inet_pton()函数转换
// 或者直接传入 INADDR_ANY 这是一个宏定义，值为0，传入之后可以是本机的任意一个IP地址
// 给套接字绑定 IP地址、端口号
int bret = bind(lfd,(struct sorkaddr *)&serAddr,sizeof(serAddr));
if(bret<0)
{
    perror("bind error");
}
```

大端存储、小端存储：数据的存储方式（目前一般是小端）

小端：高字节（高位）存在高地址，低字节（低位）存在低地址

小端：高字节（高位）存在低地址，低字节（低位）存在高地址



检验一台电脑是小端还是大端：创建一个联合体

```c
// 联合体检测大小端
union Un
{
    int a;
    char c;
}un;
int main()
{
    un.a = 1;

    if(un.c)
    {
        printf("小端\n");
    }
    else
    {
        printf("大端\n");
    }
    return 0;
}
```

hton函数族，使用 htons() 函数将短整型的端口号从主机字节序转为网络字节序

```c
uint32_t htonl(uint32_t hostlong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);
```

上面是服务器端的socket，还得有 客户端的 soctet组成socket，组成一对套接字才能进行通信

先监听

```c
// 得有一个接收方
// 监听，等着 lfd发送内容，第二个参数传一个大于0的数
listen(lfd,64);
```

再接收

使用accept()函数接收消息

函数描述

`accept` 函数是用于在服务器端接受客户端连接的系统调用

函数原型

```c
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

函数参数

（1）int sockfd

是一个已经通过 `socket` 函数创建并绑定到特定地址的监听套接字（通常是服务器的监听套接字）

（2）struct sockaddr *addr

是一个指向 `struct sockaddr` 类型的指针，用于存储连接的对端地址信息。可以为 `NULL`，表示不关心对端地址。

（3）socklen_t *addrlen

是一个指向 `socklen_t` 类型的指针，用于指定 `addr` 缓冲区的大小。在调用 `accept` 之前，`addrlen` 应该被初始化为 `struct sockaddr` 缓冲区的大小，函数返回时，`addrlen` 会被设置为实际地址结构的长度。

函数返回值

如果连接成功，返回一个新的文件描述符，这个文件描述符用于与客户端通信。

如果失败，返回 -1，并设置 `errno` 表示错误原因。

```c
struct sockaddr_in cliAddr;
socklen_t addrlen = sizeof(cliAddr);
```

```
int cfd = accept(lfd,(struct sockaddr*)&cliAddr,&addrlen);
```

因此，网络通信的流程需要 这四个函数 socket(创建)、bind(绑定)、listen(监听)、accept(接收)

```
// socket简单实现
// 自定义端口号
// 需要考虑大小端的问题，也就是网络字节序和主机的大小端是否相同（一般不同）
// 一般主机的字节序是小端存储、网络字节序是大端存储
// 因此，需要自己转换字节序
#define SOCKPORT 8002
int main(int argc, char* argv[])
{
    // 创建socket
    // 参数
    // AF_INET: 套接字协议域为IPV4协议
    // SOCT_STREAM: 套接字类型为 流套接字（安全，稳定，不丢包，对应TCP协议）
    // 0 : 套接字使用默认协议（流式套接字为TCP，报式套接字为UDP协议）
    // TPC协议参数 IPPROTO_TCP
    int lfd = socket(AF_INET,SOCK_STREAM,0);  // 创建一个流式套接字
    if(lfd<0)
    {
        perror("socket error");
    }
    int lfd1 = socket(AF_INET,SOCK_DGRAM,0);  // 创建一个报式套接字
    int lfd2 = socket(AF_INET,SOCK_RAW,0);  // 创建一个原始套接字
    // 返回一个 socket描述符
    // SOCK_STREAM 是流式协议
    // SOCK_DGRAM  是报式协议
    // TCP是流式的代表
    // UDP是报式的代表
    // 使用 bind()函数，绑定IP地址、端口号
    struct sockaddr_in serAddr;
    /*
    sockaddr_in 结构体的参数
    in_family指代协议族，在socket编程中只能是AF_INET
    sin_port存储端口号（使用网络字节顺序），2个字节（16位，0-65535）
    sin_addr存储IP地址，使用in_addr这个数据结构
    */
    serAddr.sin_family  = AF_INET;
    // 端口号一般大于1000，找个没人用的端口号
    // 一般主机的字节序是小端存储、网络字节序是大端存储
    // 因此，需要自己转换字节序
    serAddr.sin_port = htons(SOCKPORT);
    // htons 将主机字节序转为网络字节序（短整型）
    // host to network short
    // 主机 向  网络    短整型
    // ip 地址 192.168.10.128 --- 主机（本地）字节序的字符串
    // uint32_t 应该是一个32位（4个字节的）整型
    // 因此要将 上面 本机字节序的字符串 转换为 一个 网络字节序的整型
    serAddr.sin_addr.s_addr = INADDR_ANY;
    // 使用 inet_pton()函数转换
    // 或者直接传入 INADDR_ANY 这是一个宏定义，值为0，传入之后可以是本机的任意一个IP地址
    // 给套接字绑定 IP地址、端口号
    int bret = bind(lfd,(struct sockaddr *)&serAddr,sizeof(serAddr));
    if(bret<0)
    {
        perror("bind error");
    }
    // 得有一个接收方
```

```
    // 监听，等着 lfd发送内容，第二个参数传一个大于0的数
    listen(lfd,64);
    // 接收
    // int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
    struct sockaddr_in cliAddr;
    socklen_t addrlen = sizeof(cliAddr);
    int cfd = accept(lfd,(struct sockaddr*)&cliAddr,&addrlen);
    if(cfd<0)
    {
        perror("accept errot");
    }
    char buf[1024];
    // 进行通信
    while(1)
    {
        // 读取
        int read_count = read(cfd,buf,sizeof(buf));
        // 写到终端
        int write_count = write(STDOUT_FILENO,buf,read_count);
        // 再发回去
        write(cfd,buf,read_count);
    }
    return 0;
}
```

在终端 使用 nc 192.168.10.128 8001 即可实现通信

```
weihong@weihong:~/网络编程/day_626/socket$ ./tcp_server
nihao
```

```
weihong@weihong:~/网络编程/day_626/socket$ nc 192.168.10.128 8002
nihao
nihao
```